System Design Overview

System design is the art and science of creating architectures for large-scale, distributed systems. It's about building systems that can handle massive amounts of data, millions of users, and high-performance demands.

Key Concepts

- **Large-Scale Systems:** These systems handle significant volumes of data and users, like Google Maps or social media platforms.
- **Distributed Systems:** Servers are spread across multiple locations to ensure performance, reliability, and fault tolerance. For instance, users in the United States would connect to servers closer to their location for faster response times.
- **Design Patterns:** These are reusable solutions to common design problems. Examples include the Publisher-Subscriber pattern for efficient communication and the Load Balancer pattern for distributing traffic across servers.
- **Business Requirements and Data Mapping:** Engineers translate business needs into data structures and processes. In a live streaming platform, user interactions like watching videos or commenting are mapped to database objects.
- **Fault Tolerance:** Systems should be able to continue functioning even if components fail. This is achieved through redundancy and backups.
- **Extensibility:** Systems should be designed to accommodate new features easily without major overhauls. For example, a messaging app might be extended to include video calls without significant changes to the core architecture.
- **Testing:** Thorough testing is essential to ensure the system meets performance requirements and handles edge cases. Load testing, stress testing, and security testing are common practices.

Why do we need system Design

The ultimate goal of system design is to create systems that are:

- **Scalable:** They can handle increasing loads without compromising performance.
- **Reliable:** They are dependable and minimize downtime.
- **Efficient:** They utilize resources optimally.
- **Maintainable:** They are easy to understand, modify, and update.

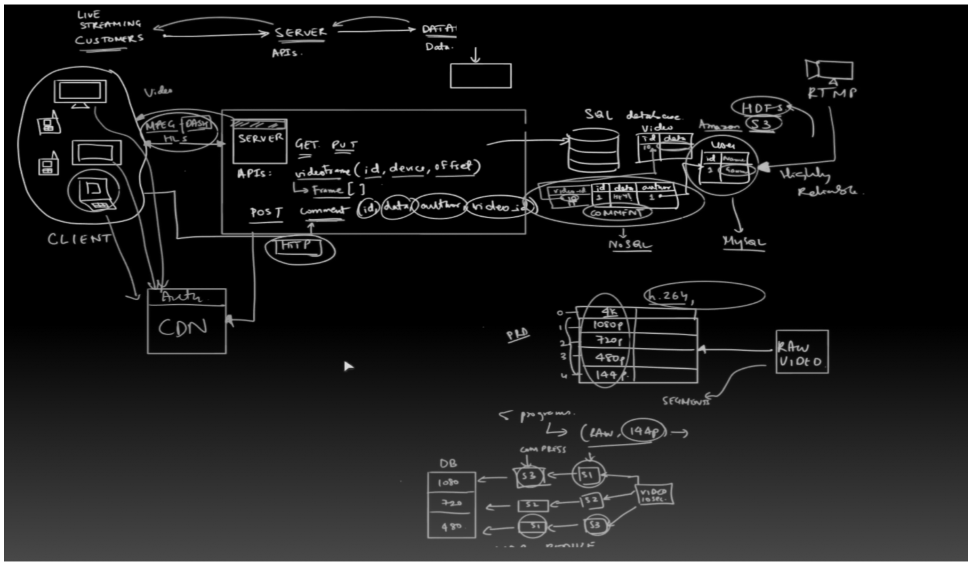
Example of a High-Level System Design for a Live Streaming Platform

Key Components:

- **Video Capture & Storage:** Capture and store videos. Compress for different devices.
- **API Design:** Create APIs for streaming, comments, and failure handling.
- **Backend:** Handle millions of users in real-time using gRPC, HTTP, and FTP.
- **Database:** Store videos, comments, and user information.
- **Design Approach:** Client-server and database-server interactions.
- **Network Protocols:** Use WebRTC for real-time streaming HTTP/HTTPS for non-real-time interactions.
- **Scalability & Resilience:** Ensure the system can handle increased loads and recover from failures.

Steps:

1. **Capture & Store:** Record videos and compress them for various devices.
2. **Design APIs:** Create APIs for streaming, comments, and handling errors.
3. **Build Backend:** Use gRPC, HTTP, and FTP to handle a large number of users.
4. **Design Database:** Store video, comment, and user data.
5. **Implement Interactions:** Use client-server and database-server interactions.
6. **Choose Protocols:** Use WebRTC for real-time streaming and HTTP/HTTPS for other interactions.
7. **Ensure Scalability & Resilience:** Use techniques like caching, load balancers, and auto-scaling.



Remember: Test the system thoroughly to identify and fix any issues before launch.

Conclusion

This architecture efficiently supports video streaming, API interactions, and data storage while ensuring scalability and resilience. Implementation details can be refined based on testing and feedback.

Introduction to Low-Level Design

Low-level design focuses on the granular details of system components and their interactions. It's a step-by-step breakdown of high-level design, addressing algorithms, data structures, and interfaces.

Contrast with High-Level Design

- **High-level design** deals with the overall system architecture, components, and their interactions.
- **Low-level design** dives into the specific implementation details of each component.

Example: Video Streaming System

- **High-level:** The system has a user interface, a video server, and a database.
- **Low-level:**
  - **User interface:** How users interact with the system (play, pause, seek).
  - **Video server:** How videos are stored, retrieved, and streamed.
  - **Database:** How user data and video metadata are stored and accessed.

Key Considerations in Low-Level Design

- **User experience:** Ensure a smooth and intuitive user experience.
- **Data structures:** Choose appropriate data structures for efficient storage and retrieval.
- **Algorithms:** Implement efficient algorithms for tasks like video processing and buffering.
- **API design:** Create well-defined APIs for communication between components.
- **Performance:** Optimize for speed, scalability, and resource utilization.
- **Fault tolerance:** Handle errors and failures gracefully.

Design Process

1. **Identify use cases:** Determine the actions users can perform.
2. **Create UML diagrams:** Use use case, class, and sequence diagrams to visualize the system.
3. **Define classes and objects:** Identify the entities and their attributes and behaviors.
4. **Design algorithms:** Develop algorithms for core functionalities.

5. **Implement APIs:** Define the interfaces for communication between components.
6. **Optimize performance:** Identify bottlenecks and improve efficiency.

Example: Video Streaming Class Diagram

- **Video:** id, title, duration, metadata
- **User:** id, name, preferences
- **VideoStreamingService:** playVideo, pauseVideo, seekVideo, getVideoFrame

Conclusion

Low-level design is crucial for building robust and efficient software systems. By carefully considering user experience, data structures, algorithms, and APIs, you can create high-quality software that meets user needs and performs well.

Basics of system design

Introduction to Load Balancers

A load balancer is a crucial component in system design that distributes incoming traffic across multiple servers to ensure optimal performance and availability. It prevents a single server from becoming overwhelmed, improving overall system responsiveness and reliability.

How Load Balancers Work

- **Multiple servers:** Load balancers manage a pool of servers (nodes) that can handle incoming requests.
- **Traffic distribution:** Incoming requests are distributed across the servers using a chosen algorithm.
- **Health checks:** Load balancers monitor the health of individual servers and remove unhealthy ones from the pool.

Types of Load Balancers

- **Hardware load balancer:** A dedicated physical appliance that handles load balancing.
- **Software load balancer:** A software application running on a server that performs load-balancing functions.

Common Load Balancing Algorithms

- **Round-robin:** Distributes requests in a circular fashion, ensuring each server receives an equal number of requests.
- **Weighted round-robin:** Prioritizes servers based on their capacity or performance.
- **Least connections:** Directs requests to the server with the fewest active connections.
- **Least response time:** Sends requests to the server that has responded to previous requests the fastest.
- **Source IP hash:** Distributes requests based on the source IP address, ensuring consistency for clients.
- **URL hash:** Distributes requests based on the URL, ensuring specific content is always served by the same server.

Where to Add Load Balancers

Load balancers can be added at various points in a system architecture:

- **In front of web servers:** To distribute incoming HTTP requests.
- **Between web servers and application servers:** To balance load for application logic.
- **In front of databases:** To distribute database queries across multiple database instances.

Benefits of Using Load Balancers

- **Improved performance:** Prevents overloading of individual servers.
- **Increased availability:** Ensures the system remains operational even if some servers fail.
- **Scalability:** Allows for easy addition or removal of servers to accommodate changing load.
- **Enhanced fault tolerance:** Protects against single points of failure.

Caching in System Design

Caching is a fundamental optimization technique used to improve system performance and reduce load. It leverages the principle of locality of reference, which states that data recently accessed is likely to be accessed again soon.

Caching as Short-Term Memory

Imagine a cache as your short-term memory. It holds a limited amount of data that you've recently used, allowing for faster retrieval compared to accessing long-term memory (the main storage).

Cache Placement

Caches can be implemented at various levels in a system architecture:

- **Hardware:** CPU cache stores frequently accessed instructions and data for the processor.
- **Operating System:** OS caches file system access to speed up disk reads.
- **Web Browsers:** Browsers cache web pages and static content to minimize reloading from the server.
- **Web Applications:** Application-level caches store frequently accessed database queries or API responses.

Types of Caches

1. **Application Server Cache:**
    a. Stores responses on the request layer node, enabling faster delivery if the data is already cached.
    b. Challenge: With load balancers distributing requests across multiple nodes, the same request might go to different nodes, causing cache misses (data not found).
    c. Solutions:
        i. **Distributed Cache:** Divides the cache among nodes using consistent hashing. Each node knows where to look for specific data within the distributed cache. This allows easy scaling by adding more nodes.
        ii. **Global Cache:** All nodes share a single cache space. Each node queries the cache like a local one.
            1. There are two options for handling cache misses:
                a. The cache retrieves the missing data. (Suitable for low cache miss rates)
                b. Request node retrieves the missing data. (Suitable for high cache hit rates)

2. **Content Delivery Network (CDN):**
    a. A network of geographically distributed servers that cache static content (images, videos, etc.) for faster delivery to users.
    b. Reduces origin server load and improves user experience, especially for geographically distant users.
    c. Alternative for smaller systems: Serve static content from a separate subdomain using a lightweight server like Nginx.

Cache Invalidation

When data is modified in the original source (e.g., database), the corresponding cached data must be invalidated to ensure consistency.

- **Write-Through Cache:** Updates the cache and the original source simultaneously. Ensures data consistency but has higher write latency.
- **Write-Around Cache:** Bypasses the cache and writes directly to the original source. Reduces write operations on the cache but may lead to cache misses for recently written data.
- **Write-Back Cache:** Writes data to the cache first and updates the original source asynchronously (at intervals). Provides low latency and high throughput but risks data loss in case of a system crash.

Cache Eviction Policy

Caches have limited space, so an eviction policy determines which data to remove when the cache reaches capacity. Here are some common policies:

- **FIFO (First-In-First-Out):** Removes the oldest data (least recently used). Simple to implement but might not be optimal for frequently accessed older data.
- **LIFO (Last-In-First-Out):** Removes the newest data. Might not be ideal if new data is less frequently accessed.
- **LRU (Least Recently Used):** Evicts the data that hasn't been accessed for the longest time. Often a good choice for caching as it prioritizes recently used data. This can be implemented using a Doubly Linked List and a hash table for efficient lookups.
- **MRU (Most Recently Used):** Opposite of LRU, removes the most recently accessed data. Less common but might be useful in specific scenarios.
- **LFU (Least Frequently Used):** Removes the data that has been accessed the least number of times. More complex to implement but can be beneficial for data with varying access frequencies.
- **Random Replacement:** Select a random data item for eviction. Simple but might not be the most efficient strategy.

Sharding in System Design

Sharding is a technique used to partition large datasets across multiple databases or servers to improve scalability and performance. It's essential for handling massive amounts of data that a single database cannot efficiently manage.

- **Data Partitioning:** Dividing a large dataset into smaller, more manageable chunks.
- **Horizontal Sharding:** Partitioning data based on ranges or values of specific attributes
- **Vertical Sharding:** Partitioning data based on different types of data
- **Directory-Based Sharding:** Assigning each data item to a specific shard based on a directory or mapping function.

Benefits of Sharding

- **Scalability:** Easily add or remove shards to accommodate increasing or decreasing data loads.
- **Performance:** Reduces the load on individual databases, improving query performance.
- **Fault Tolerance:** Improves system resilience by distributing data across multiple nodes.
- **Data Availability:** Increases data availability by reducing the risk of a single point of failure.

Challenges of Sharding

- **Distributed Systems Complexity:** Managing distributed systems can be complex, especially when ensuring data consistency across multiple shards.
- **Join Queries:** Joining data across multiple shards can be inefficient due to network communication and data transfer.
- **ACID Compliance:** Maintaining ACID (Atomicity, Consistency, Isolation, Durability) properties can be challenging in distributed shared systems.

Sharding Criteria

- **Hash-Based Sharding:** Assigns data to shards based on a hash function applied to a specific attribute (e.g., user ID).
- **List Partitioning:** Divides data into shards based on predefined lists or ranges (e.g., geographic regions).
- **Round-Robin Partitioning:** Distributes data evenly across shards in a circular fashion.
- **Composite Partitioning:** Combines multiple sharding strategies (e.g., horizontal and vertical).

Choosing the Right Sharding Strategy

- **Data Distribution:** How evenly is data distributed across the dataset?
- **Query Patterns:** What types of queries are commonly performed?
- **Data Consistency Requirements:** How important is ACID compliance?
- **Scalability Needs:** How much growth is expected in the future?

Indexes in System Design

Indexes are a fundamental optimization technique used in databases to improve the speed of data retrieval operations significantly

. They function like a table of contents in a book, pointing to the exact location of the data you need, minimizing search time.

Understanding the Trade-off

While indexes offer significant benefits for reads, they come with a trade-off:

- **Increased Storage Overhead:** Indexes themselves require additional storage space.
- **Slower Writes:** Maintaining indexes adds a small overhead to write operations, as updates to indexed columns need to be reflected in the index structure.

How Indexes Work

Imagine an index as a separate data structure that stores a copy of specific column values along with pointers to the corresponding rows in the main table. When you query the database using these indexed columns, it can quickly locate the relevant data by searching the index instead of scanning the entire table.

Types of Indexing:

Databases typically support two main types of indexing:

- **Ordered Indexing:** This type stores the indexed column values in sorted order (ascending or descending). It's ideal for range queries where you search for data within a specific range of values.
- **Hash Indexing:** This type uses a hash function to map column values to unique hash values. Hash indexes are faster for equality comparisons (exact matches) but less efficient for range queries.

Proxies in system design

Proxies offer a multitude of functionalities that enhance network security, performance, and efficiency. Here are some key capabilities:

- **Filtering Requests:** Proxies can filter out unwanted requests, such as malicious content or access attempts to restricted websites.
- **Logging Requests:** They can maintain a log of all requests and responses, which can be helpful for security audits and troubleshooting.
- **Transforming Requests:** Proxies can modify requests by adding or removing headers, encrypting or decrypting data, or compressing content to optimize bandwidth usage.
- **Caching:** Proxies can cache frequently accessed data, reducing load on the origin server and improving response times for clients.
- **Load Balancing:** They can distribute incoming requests across multiple servers, ensuring better performance under high traffic conditions.
- **Request Optimization:** Some proxies can combine similar requests from multiple clients into a single request to the server, reducing overall network traffic.
- **Spatial Locality Optimization:** For geographically distributed data storage, proxies can group requests for nearby data, minimizing retrieval latency.

Benefits of Using Proxies

The advantages of employing proxies are numerous:

- **Improved Security:** Filtering capabilities help shield your network from malicious activities.
- **Enhanced Performance:** Caching and load balancing lead to faster response times and a smoother user experience.
- **Reduced Server Load:** Proxies take pressure off the origin server by handling tasks like filtering and caching.
- **Optimized Traffic Flow:** Request merging and spatial optimization improve network efficiency.

Proxies are particularly useful in scenarios with:

- **High Traffic Load:** Proxies can handle large volumes of requests, ensuring system stability.
- **Limited Caching:** If your application has limited caching capabilities, a proxy can provide significant performance benefits.
- **Security Concerns:** Proxies add an extra layer of protection for your network by filtering and monitoring traffic.

Queues in System Design

Queues facilitate asynchronous communication and reliable message processing. They play a crucial role in decoupling systems, ensuring scalability, and handling high volumes of data.

Understanding Queues

A queue is a data structure that stores messages in a first-in, first-out (FIFO) order. Messages are placed at the end of the queue and are processed one at a time, ensuring fair and orderly execution.

How Message Queues Work

1. **Producer:** A producer generates a message and sends it to a queue.
2. **Broker:** A message broker is a software component that manages the queue. It receives messages from producers and stores them in the queue.
3. **Consumer:** A consumer subscribes to a queue and receives messages from it. It processes each message individually and then acknowledges its completion.

Benefits of Using Queues

- **Decoupling:** Queues decouple producers and consumers, allowing them to operate independently without direct communication. This improves system flexibility and scalability.
- **Asynchronous Processing:** Producers can send messages and continue with their work without waiting for a response, while consumers can process messages at their own pace.
- **Load Balancing:** Queues can distribute messages across multiple consumers, ensuring efficient workload distribution and preventing bottlenecks.
- **Fault Tolerance:** If a consumer fails or becomes unavailable, messages can be re-queued and processed by another consumer, ensuring fault tolerance.
- **Peak Handling:** Queues can buffer messages during peak loads and process them gradually when the load subsides, preventing system overload.

SQL vs. NoSQL Databases

SQL Databases

- **Structure:** Organized in rows and columns with a predefined schema.
- **Data Type:** Primarily handles structured data.
- **ACID Compliance:** Ensures Atomicity, Consistency, Isolation, and Durability of transactions.

- **Scalability:** Primarily vertically scalable (increasing hardware resources).
- **Use Cases:**
    - Applications with well-defined schemas and complex relationships.
    - Data integrity and consistency are critical.
    - Legacy systems or applications that require SQL for querying.

NoSQL Databases

- **Structure:** Flexible schemas that can adapt to changing data structures.
- **Data Type:** Handles structured, semi-structured, and unstructured data.
- **ACID Compliance:** May not guarantee full ACID compliance, depending on the specific database.
- **Scalability:** Highly horizontally scalable (adding more nodes).
- **Types:**
    - **Key-Value Stores:** Store data as key-value pairs. Examples: Redis, Voldemort, DynamoDB.
    - **Document Databases:** Store data in JSON or BSON documents. Examples: MongoDB, CouchDB.
    - **Wide-Column Databases:** Store data in columns, with each column representing a family of values. Examples: Cassandra, HBase.
    - **Graph Databases:** Store data as nodes (entities) connected by edges (relationships). Examples: Neo4j, InfiniteGraph.

When to Choose SQL:

- **Well-defined schema:** If your data has a clear structure and is unlikely to change significantly.
- **ACID compliance:** If data integrity and consistency are paramount.
- **Complex queries:** If you need to perform complex join operations or aggregations.

When to Choose NoSQL:

- **Flexible schema:** If your data structure is evolving or uncertain.
- **Large datasets:** If you're dealing with massive amounts of data that require horizontal scaling.
- **High performance:** If you need low-latency access to data and can sacrifice some ACID guarantees.
- **Specific data models:** For graph-structured data (social networks, recommendation systems) or time-series data (IoT, financial data).

Monolithic vs. Microservices Architecture

Monolithic Architecture

- **Description:** A single, self-contained unit containing all application components.
- **Benefits:**
  - Easier to develop and manage initially.
  - Simple deployment and testing.
- **Drawbacks:**
  - Tightly coupled components make it difficult to scale or modify independently.
  - Limited agility and flexibility.
  - Single point of failure risk.

Microservices Architecture

- **Description:** A collection of small, independent services that work together to form an application.
- **Benefits:**
  - **Scalability:** Each service can be scaled independently based on demand.
  - **Fault tolerance:** Failure of one service doesn't affect the entire application.
  - **Agility:** Easier to add, remove, or modify individual services.
  - **Continuous delivery:** Enables faster development and deployment cycles.
  - **Technology heterogeneity:** Different services can use different technologies and frameworks.
- **Drawbacks:**
  - Increased complexity due to multiple services.
  - Requires careful coordination and management of inter-service communication.
  - Potential for increased overhead due to network communication.

Best Practices for Microservices

- **Separate data stores:** Use separate databases for each service to avoid coupling and improve scalability.
- **Separate by features:** Group related functionalities into a single service.
- **Stateless servers:** Design services to be stateless, making them easier to scale and manage.

REST APIs: A Building Block for Modern Systems

REST APIs (Representational State Transfer APIs) have become the de facto standard for building web services and APIs. They provide a simple, scalable, and stateless way for applications to communicate over HTTP.

Key Characteristics of REST APIs

- **Stateless:** Each request is treated independently, without relying on previous requests. All necessary information is included in the request itself.
- **Cachable:** Responses can be cached to improve performance and reduce server load.
- **Layered System:** Supports multiple layers of abstraction, allowing for flexibility and scalability.
- **Client-Server:** Separates concerns between the client (making requests) and the server (handling requests and data).

Benefits of REST APIs

- **Simplicity:** RESTful APIs are easy to understand and implement.
- **Scalability:** They can handle large volumes of traffic and scale horizontally.
- **Flexibility:** REST APIs support various data formats (JSON, XML, plain text) and can be used with different programming languages and platforms.
- **Efficiency:** REST APIs are often more efficient than SOAP-based APIs due to their lightweight nature.

Creating REST APIs

You can create REST APIs using various programming languages and frameworks:

- **Java:** Spring Boot, Dropwizard, Jersey
- **Node.js:** Express.js, Koa
- **Python:** Flask, Django REST framework
- **Ruby:** Sinatra, Rails

Hashing in System Design

Hashing is a fundamental technique used in computer science to efficiently store and retrieve data. It involves transforming a data item (e.g., a string or number) into a fixed-size hash value using a mathematical function. This hash value is then used as an index to locate the data in a hash table.

Purpose of Hashing

Hashing serves several key purposes in system design:

- **Indexing data:** Hashing is used to create indexes for databases, allowing for rapid data retrieval based on specific keys.
- **Cryptography:** Hashing functions are used to create cryptographic hashes for password storage and data integrity verification.
- **Data deduplication:** Hashing can be used to identify duplicate data elements within a dataset.
- **Distributed systems:** Hashing is essential for distributing data across multiple nodes in a distributed system.

Hash Function and Hash Table

- **Hash Function:** A mathematical function that maps data of arbitrary size to a fixed-size hash value. Good hash functions should produce a uniform distribution of hash values and minimize collisions.
- **Hash Table:** A data structure that stores key-value pairs using hashing. The hash function is used to map the key to an index within the hash table, where the corresponding value is stored.

Collision Handling

When multiple data items hash to the same index, a collision occurs. There are several techniques to handle collisions:

- **Separate Chaining:** Store colliding items in a linked list associated with the hash table index.
- **Linear Probing:** Search for the next available slot in the hash table and store the item there.
- **Quadratic Probing:** Use a quadratic probing function to determine the next slot to search.
- **Double Hashing:** Use a second hash function to calculate the step size for probing.

Kafka: A Distributed Streaming Platform

Kafka is a high-performance, distributed streaming platform that is widely used for building real-time data pipelines and applications. It is designed to handle massive volumes of data and provides features like fault tolerance, scalability, and durability.

Key Components of Kafka

- **Producers:** Applications that generate data and publish it to Kafka topics.
- **Topics:** Logical categories for organizing data streams.
- **Brokers:** Servers that store and process messages.
- **Consumers:** Applications that subscribe to topics and consume messages.
- **Partitions:** Subdivisions of a topic that distribute data across multiple brokers for scalability and fault tolerance.

Why Use Kafka?

- **Scalability:** Kafka can handle massive volumes of data and scale horizontally by adding more brokers.
- **Fault tolerance:** Kafka replicates data across multiple brokers to ensure durability and availability.
- **Durability:** Messages are persisted to disk, guaranteeing that they won't be lost even if brokers fail.
- **Real-time processing:** Kafka enables real-time processing of data streams, making it suitable for applications like IoT, financial data processing, and recommendation systems.
- **Decoupling:** Kafka decouples producers and consumers, allowing them to operate independently and at different rates.

Kafka Use Cases

- **Real-time data pipelines:** Processing and analyzing data streams in real time.
- **Message queuing:** Asynchronous communication between microservices.
- **Log aggregation:** Collecting and analyzing logs from distributed systems.
- **Streaming analytics:** Performing analytics on real-time data streams.
- **IoT data processing:** Handling large volumes of data generated by IoT devices.

Kafka vs. Other Message Queues

- **Distributed:** Kafka is designed for distributed systems and can handle large-scale data processing.
- **Durability:** Kafka guarantees message durability through replication and persistence.
- **Scalability:** Kafka can easily scale horizontally by adding more brokers.
- **Real-time processing:** Kafka is optimized for real-time data processing.

LRU Cache Eviction Policy

LRU (Least Recently Used) is a popular cache eviction policy that removes the least recently accessed items from the cache when they reach capacity. It's often considered the most effective cache eviction strategy, especially for systems with limited cache space.

How LRU Works

1. **Doubly Linked List:** A doubly linked list is used to maintain the order of items in the cache. The most recently used items are at the head of the list, while the least recently used items are at the tail.
2. **Hash Map:** A hash map is used to quickly locate items in the cache based on their keys.
3. **Cache Capacity:** The cache has a predefined size limit.
4. **Item Access:** When an item is accessed, it's moved to the head of the linked list
5. **Cache Full:** If the cache is full and a new item needs to be added, the least recently used item (at the tail of the linked list) is removed.

Implementation

To implement LRU, you can use a doubly linked list and a hash map:

1. **Create a doubly linked list node:** Each node represents a cached item and contains a key, value, and pointers to the previous and next nodes.
2. **Create a hash map:** The hash map stores the key-value pairs and references to the corresponding nodes in the linked list.
3. **Access an item:**
   a. Check if the item exists in the hash map.
   b. If it exists, remove it from the linked list and insert it at the head.
   c. If it doesn't exist, add it to the linked list and hash map.
4. **Evict an item:** If the cache is full, remove the node at the tail of the linked list and remove its corresponding entry from the hash map.

Advantages of LRU

- **Simplicity:** LRU is relatively easy to implement.
- **Efficiency:** It provides good performance in most cases.
- **Adaptability:** LRU adapts well to changing access patterns.

Disadvantages of LRU

- **Potential for thrashing:** In some scenarios, LRU can cause thrashing, where frequently accessed items are evicted and then immediately reloaded.
- **Bias towards recent accesses:** LRU may not be optimal for applications with non-uniform access patterns.

Apache Hadoop

Apache Hadoop is an open-source framework designed to process massive amounts of data across clusters of commodity hardware. It's widely used for big data analytics, data warehousing, and machine learning.

Core Components

- **Hadoop Distributed File System (HDFS):** A distributed file system that stores data across multiple nodes.
- **MapReduce:** A programming model for processing large datasets in parallel.
- **Yarn:** A resource management framework that allocates resources to applications.

How Hadoop Works

1. **Data Ingestion:** Data is ingested into HDFS, which splits it into large blocks and distributes them across multiple nodes.
2. **MapReduce Job Submission:** A MapReduce job is submitted to the Yarn resource manager.
3. **Task Scheduling:** Yarn schedules the MapReduce tasks across the cluster, taking into account data locality and resource availability.
4. **Map Phase:** Map tasks process individual data blocks in parallel, emitting key-value pairs.
5. **Shuffle and Sort:** The intermediate key-value pairs are shuffled and sorted based on the keys.
6. **Reduce Phase:** Reduce tasks combine the intermediate values for each key, producing the final output.

Key Features of Hadoop

- **Scalability:** Hadoop can handle massive datasets by distributing the workload across multiple nodes.
- **Fault Tolerance:** It is designed to handle hardware failures and automatically recover lost data.
- **Flexibility:** Hadoop supports a variety of data formats and programming languages.
- **Cost-effectiveness:** It can be deployed on low-cost commodity hardware.

HDFS

- **Distributed Storage:** HDFS stores data across multiple nodes for redundancy and scalability.
- **Replication:** Data is replicated to ensure fault tolerance and availability.
- **Rack Awareness:** HDFS is aware of the physical layout of the cluster and replicates data across different racks to improve fault tolerance.

Apache HBase

Apache HBase is a distributed, column-oriented NoSQL database that is designed to handle massive amounts of data. It is built on top of the Hadoop Distributed File System (HDFS) and is widely used in big data applications.

Key Features of HBase

- **Scalability:** HBase can scale horizontally to handle large datasets by adding more nodes to the cluster.
- **Durability:** Data is replicated across multiple nodes to ensure fault tolerance and durability.
- **Performance:** HBase is optimized for fast read and write operations, making it suitable for real-time applications.
- **Column-oriented:** Data is organized into columns, which can be indexed and queried independently.
- **Sparse data handling:** HBase is designed to efficiently handle sparse data, where most values in a column are null.

HBase Architecture

- **ZooKeeper:** A distributed coordination service that manages the HBase cluster.
- **HMaster:** The master server that coordinates the region servers and assigns regions to them.
- **Region Servers:** Processes client requests and store data in regions.
- **Regions:** Subdivisions of a table that are stored on individual region servers.
- **HDFS:** The underlying distributed file system that stores HBase data.

HBase Use Cases

- **Real-time analytics:** Processing large volumes of data in real time.
- **Time series data:** Storing and querying time-series data.
- **Internet of Things (IoT):** Processing data from IoT devices.
- **Clickstream analysis:** Analyzing user behavior data.

ZooKeeper: A Distributed Coordination Service

ZooKeeper is a distributed coordination service that provides a range of features for managing and coordinating distributed applications. It's commonly used in distributed systems to ensure consistency, reliability, and scalability.

Key Functions of ZooKeeper

- **Configuration Management:** Stores and distributes configuration information across a cluster of machines.
- **Naming Service:** Provides a hierarchical naming system for naming and addressing nodes in a distributed system.
- **Synchronization:** Offers primitives like locks, barriers, and queues for coordinating distributed processes.
- **Group Services:** Facilitates leader election and group membership management.

ZooKeeper Architecture

- **Cluster:** A ZooKeeper ensemble consists of multiple nodes (servers).
- **Leader Election:** One node acts as the leader, coordinating the cluster and handling updates.
- **Replication:** All nodes maintain a copy of the ZooKeeper state, ensuring consistency.
- **Clients:** Applications connect to any ZooKeeper node to access its services.

ZooKeeper Data Model

- **ZNodes:** The basic unit of data in ZooKeeper. They can be directories or files.
- **Hierarchical Structure:** ZNodes are organized in a hierarchical structure, similar to a file system.
- **Data:** ZNodes can store data, which can be accessed and modified by clients.

ZooKeep offers

- **Sequential Consistency:** All clients see the same sequence of changes to ZooKeeper data.
- **Atomicity:** Updates to ZooKeeper data are atomic, ensuring that they either succeed or fail as a whole.

Apache Solr: A Powerful Full-Text Search Engine

Apache Solr is an open-source enterprise search platform that provides a highly scalable and efficient way to search, index, and analyze large datasets. It's built on top of the Apache Lucene search engine and offers a rich set of features for building robust search applications.

Key Features of Solr

- **Full-text search:** Solr can search for any text within a document, making it ideal for applications that require advanced search capabilities.
- **Faceting:** Allows users to filter search results based on different criteria, such as categories, price ranges, or attributes.
- **Real-time indexing:** Solr can index documents in real time, ensuring that search results are always up-to-date.
- **Scalability:** Solr can scale horizontally to handle large datasets and high traffic loads.
- **Integration:** Solr integrates seamlessly with other Apache Hadoop components, such as HDFS and MapReduce.
- **Rich API:** Solr provides a RESTful API and Java API for interacting with the search engine.

Use Cases for Solr

- **E-commerce search:** Providing fast and accurate product search for online stores.
- **Enterprise search:** Searching for documents, files, and other content within an organization.
- **Content management systems:** Indexing and searching large volumes of content.
- **Analytics:** Analyzing search queries and user behavior to improve search results.

Solr Architecture

- **SolrCloud:** A distributed architecture that allows Solr to scale horizontally and handle high-traffic loads.
- **ZooKeeper:** Used for coordination and configuration management.
- **Collections:** Logical containers for storing documents.
- **Shards:** Subdivisions of a collection that are distributed across multiple nodes.
- **Replicated shards:** Multiple copies of a shard stored on different nodes for redundancy.

Cassandra: A Scalable, Distributed NoSQL Database

Apache Cassandra is a highly scalable, distributed NoSQL database designed to handle massive amounts of data. It's known for its fault tolerance, high availability, and performance.

Key Features of Cassandra

- **Distributed:** Cassandra is a distributed database that can be spread across multiple data centers.
- **Decentralized:** There is no single point of failure, as each node in the cluster can handle read and write operations.
- **Scalability:** Cassandra can scale horizontally to handle increasing workloads.
- **Durability:** Data is replicated across multiple nodes to ensure durability and fault tolerance.
- **Performance:** Cassandra is optimized for high performance and can handle large amounts of data.
- **Column-oriented:** Data is stored in columns, making it efficient for querying specific data points.
- **Time-series data:** Cassandra is well-suited for storing and querying time-series data.

Use Cases for Cassandra

- **Real-time analytics:** Processing large volumes of data in real time.
- **Internet of Things (IoT):** Handling data from IoT devices.
- **Financial data:** Storing and analyzing financial data.
- **Content management systems:** Managing large amounts of content.

Cassandra Architecture

- **Nodes:** Cassandra is made up of multiple nodes that store and replicate data.
- **Clusters:** A cluster is a group of nodes that work together to store and manage data.
- **Data Centers:** Cassandra can be deployed across multiple data centers for improved availability and disaster recovery.
- **Keyspaces:** Logical containers for organizing data.
- **Tables:** Tables are used to store data within keyspaces.
- **Columns:** Columns are used to store individual data points.

Designing a URL Shortener System

Requirements

- **URL Shortening:** Users should be able to create shortened URLs from original URLs.
- **Redirection:** Clicking on a shortened URL should redirect the user to the original URL.
- **Custom URLs:** Users should be able to create custom shortened URLs.
- **Expiration:** Users should be able to set an expiration time for shortened URLs.

API Creation

- **GET /shorten:** Creates a shortened URL for a given original URL.
- **GET /{shortened_url}**: Redirects to the original URL associated with the shortened URL.

Logic to Solve the Problem

1. **Generate Shortened URL:** Use a unique ID generator (e.g., UUID) to create a shortened URL.
2. **Store in Database:** Store the original URL, shortened URL, and expiration date in the database.
3. **Redirection:** When a shortened URL is accessed, retrieve the original URL from the database and redirect the user.
4. **Expiration Handling:** Check the expiration date before redirecting. If expired, return a 404 error.

System Workflow

1. The user requests a shortened URL.
2. The system generates a shortened URL and stores it in the database.
3. The shortened URL is returned to the user.
4. When a user clicks on a shortened URL, the system retrieves the original URL from the database.
5. The system redirects the user to the original URL.

HTTP VS HTTPS

How HTTP Works

- When you enter `http://` in the address bar, you're instructing your browser to establish a connection using the Hypertext Transfer Protocol (HTTP).
- HTTP primarily uses TCP (Transmission Control Protocol) over port 80 to send and receive data packets across the web.
- HTTP is a stateless protocol, meaning each request is treated independently, and the server doesn't maintain any session information.

How HTTPS Works

- HTTPS stands for Hypertext Transfer Protocol Secure. It's essentially HTTP with an added layer of security provided by Transport Layer Security (TLS).
- When you enter `https://` in the address bar, your browser establishes a secure connection using HTTPS.
- HTTPS also uses TCP to send and receive data, but it does so over port 443.
- The key difference is that HTTPS encrypts data using TLS, ensuring that it remains confidential during transmission.

How HTTPS is Secure

- HTTPS uses a combination of encryption and authentication to establish a secure connection.
- **Public Key Infrastructure (PKI):** A public key is generated and shared publicly, while a private key is kept secret.
- **SSL/TLS Certificates:** A trusted certificate authority (CA) issues an SSL/TLS certificate that binds a domain name to a public key.
- **Encryption:** When a client connects to an HTTPS server, the server presents its SSL/TLS certificate. The client verifies the certificate's authenticity and uses the public key to encrypt data.
- **Decryption:** The server uses its private key to decrypt the encrypted data.

Additional Considerations

- **Mixed Content:** Be cautious of websites that mix HTTP and HTTPS content, as this can compromise security.
- **Certificate Renewal:** Ensure that SSL/TLS certificates are renewed regularly to maintain security.
- **Browser Compatibility:** Different browsers may have varying levels of support for HTTPS features.

CAP Theorem: Consistency, Availability, Partition Tolerance

CAP Theorem is a fundamental principle in distributed systems that states that it is impossible for a distributed system to simultaneously guarantee all three of the following properties:

- **Consistency:** All nodes in the system see the same data at the same time.
- **Availability:** The system is always available for read and write operations.
-
- **Partition Tolerance:** The system remains operational even in the presence of network partitions.

Understanding CAP

- **Consistency:** Ensures that all nodes in the system have an up-to-date view of the data. This is crucial for applications where data integrity is critical.
- **Availability:** The system should always be available for read and write operations, even if some nodes are down or there are network partitions.
- **Partition Tolerance:** The system should continue to operate even if there are network partitions or communication failures between nodes.

Choosing Two Out of Three

The CAP theorem states that you can only choose two out of these three properties for your distributed system. Here are the common trade-offs:

- **CA (Consistency and Availability):** If you prioritize consistency and availability, you may sacrifice partition tolerance. This means that the system may not be able to handle network partitions, leading to potential data inconsistencies.
- **CP (Consistency and Partition Tolerance):** If you prioritize consistency and partition tolerance, you may sacrifice availability. This means that the system may become unavailable during network partitions.
- **AP (Availability and Partition Tolerance):** If you prioritize availability and partition tolerance, you may sacrifice consistency. This means that there may be temporary inconsistencies in the data during network partitions.

Examples

- **Database Systems:** Many database systems prioritize consistency and partition tolerance, which means they may sacrifice availability during network failures.
- **Content Delivery Networks (CDNs):** CDNs prioritize availability and partition tolerance, which means they may have temporary inconsistencies in data across different nodes.

Considerations for System Design

When designing a distributed system, carefully consider the trade-offs between consistency, availability, and partition tolerance based on the specific requirements of your application. For example, if data consistency is critical, you may need to sacrifice some availability during network failures. On the other hand, if availability is paramount, you may need to accept potential inconsistencies during network partitions.